REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
08-06-2007	SBIR Phase I Final Report	15 AUG 2006 – 13 JUN 2007

4. TITLE AND SUBTITLE
AppMon: Application Monitors for Not-Yet-Trusted Software

5a. CONTRACT NUMBER
FA9550-06-C-0098

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Carla Marceau

Dexter Kozen

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Odyssey Research Associates, Inc.
33 Thornwood Drive, Suite 500
Ithaca, NY 14850

8. PERFORMING ORGANIZATION REPORT

TR07-0005

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AF Office of Scientific Research
875 North Randolph Stree, RM 3112
Arlington, Virginia 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR/PKR3

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

AFRL-SR-AR-TR-07-0302

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Report developed under STTR contract for topic OSD06-SP2. AppMon represents a novel approach to monitoring the behavior of not-yet-trusted applications that avoids the disadvantages of current approaches. It is based on a *self-customizing monitor* that constrains the application's use of computer resources. A self-customizing monitor learns how the application normally uses computer resources and does not interfere with normal use. However, when the application uses resources in an unusual way, AppMon prevents potentially harmful accesses. Self-customizing monitors satisfy three important requirements on application security monitors. First, the application can be run immediately without testing or training. Second, customization is automatic, so only minimal demands are made on the user and system administrator. Finally, the self-customizing monitors are applicable to a wide variety of applications, including those that read and write files, read and write registry keys, invoke other processes, and use the Internet.

15. SUBJECT TERMS
STTR Report

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Richard A. Smith
U	U	U	SAR	28	19b. TELEPHONE NUMBER *(include area code)* (607) 257-1975

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

20070828127

Table of Contents

Table of Figures

1 Introduction

In the age of the Internet, military and other organizations need to protect critical information stored on their computers from attacks that attempt to modify system software, change or delete critical information, or steal confidential information. Yet in today's interconnected world, the typical computer runs software whose safety and correctness is far from certain. "Free" software and Internet scripts downloaded to client computers may be cleverly crafted to attract customers and deliver malware. COTS and GOTS applications may contain "Easter eggs" inserted by a malicious developer. Novel, updated, or untested applications may contain undiscovered vulnerabilities or errors that can compromise sensitive computer data.

Various technologies have been developed to enable users to run new applications with minimal risk, however, each of the current solutions lacks at least one of the essential requirements for such a technology: speed, ease of use, and widespread applicability. *Exhaustive testing* is slow. *Sandboxing* defines one-size-fits-all limits on application behavior and rules out any application that does not fit its constraints; a sandbox that prevents writing to files cannot be used with a new application that creates or modifies local files. *User-specified sandboxes* require users to state in advance what an application may or may not do with resources. Running the application in a virtual machine with shadow resources prevents harm to the computer; however, this prevents the application from having any real effect at all. The introduction of *pseudo-users* (specially defined identities with limited access rights) requires a system administrator to correctly define all access rights of the pseudo-user. *Mandatory access control* (MAC) techniques, used in LOMAC [Fraser00] and SE Linux [SELinux],[1] require the administrator to specify security levels for all resources, independent of the application—a tedious and error-prone activity.

To enable military personnel to take advantage of new software quickly, a new approach is needed that imposes little or no delay, is usable by system administrators with only modest security training, and accommodates a very wide variety of applications. In this Phase I STTR effort, ATC-NY and Cornell University have investigated just such an approach and established its feasibility.

1.1 Innovation

ATC-NY has developed a novel approach to monitoring the behavior of not-yet-trusted applications that avoids the disadvantages of current approaches. It is based on a *self-customizing monitor* that constrains the application's use of computer resources. A self-customizing monitor learns how the application normally uses computer resources and does not interfere with normal use. However, when the application uses resources in an unusual way, a sandbox prevents potentially harmful accesses. Self-customizing monitors satisfy all the requirements on application security monitors mentioned above:

---

[1] Section 9 provides references for citations throughout this proposal.

- The application can be run immediately without testing or training;
- Customization is automatic, so only minimal demands are made on the user and system administrator; and
- The self-customizing monitors are applicable to a wide variety of applications, including those that read and write files, read and write registry keys, invoke other processes, and use the Internet.

In research and experimentation carried out in Phase I, we have shown that monitors with these characteristics can in fact be constructed. We have built application profiles and simulated monitor actions with a variety of applications, including Microsoft® Office Outlook®, a notoriously complex and resource-intensive application.

The key innovation is that sandbox constraints are learned from the way the application uses computer resources, rather than dictated equally for all applications ("one size fits all") or left to human guesswork. This innovation is made possible by **empirical privilege profiling**, an automatic technique pioneered by the proposed principal investigator [Marceau05] that identifies how resources are used by an application. A naïve approach might be to simply list, for example, the files used by an application during execution; however, such an approach fails to abstract away from actual files to concepts such as "the log file" or "the file being edited." For example, if an application writes to a user-defined file, the naïve approach cannot recognize the write operation as part of the application's normal behavior. It may know that a certain file has been written that was never written before, but it has no way to decide whether the new file is "the file being edited" or the target of an attacker. By contrast, empirical privilege profiling can identify the write operation as one the application normally performs, even if that particular file has never been accessed by the application before. A self-customizing monitor based on empirical privilege profiling can thus distinguish between behavior it recognizes as normal—and therefore low-risk even though applied to a new resource—and novel behavior, which it may treat more warily.

The second innovation is an improved profiling technique that is more robust and broadly applicable than that used in the earlier work. As described in [Marceau05], empirical privilege profiling depends on knowing the location where control left an application binary and (possibly through various library functions) resulted in a call to a kernel operation. Further, this technique is not applicable to programs written in interpreted languages—such as Java or languages compiled with .Net—since for such applications, the binary executable is the interpreter. Hence, for AppMon, we are developing a new technique that models an application's use of resources based only on its calls to the kernel.

Based on these two innovations, which make it possible to automatically construct self-customizing monitors, ATC-NY is developing AppMon, an application monitor that will enable military and other users to gain the advantage of running new applications quickly while minimizing the risks.

1.2 Why preventing harm is hard

The problem that AppMon addresses is very hard to solve. First of all, since it must detect novel misbehavior, its job is similar to that of anomaly-based intrusion detection systems. That is, rather than detecting certain attacks, it has a notion of normal program behavior and detects significant and possibly harmful differences from that behavior.

For any questionable action, the question is whether to raise an alarm. Anomaly detection systems run into two kinds of problems: false positives (false alarms) and false negatives (misses). If the system is too permissive, it will fail to catch attacks. If it is too strict, it will generate many false alarms, which is actually worse, because people will first loathe it and then ignore all of the alarms, even the true ones. There is an intrinsic relation between false positives, false negatives, and the amount of information available to the detection system, as illustrated in Figure 1.[2] The various curves in Figure 1 represent increasing levels of information available to the system that must decide whether it should raise an alarm for a given event. The topmost curve represents a fairly low level of information. The star in Figure 1 represents a given false positive rate, false negative rate, and level of available information.

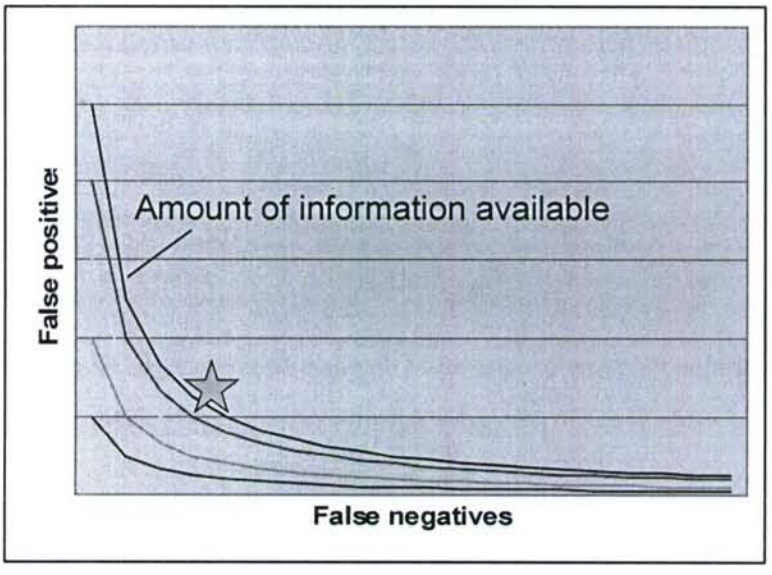

Figure 1. Avoiding harm without overburdening the user

Note that the false-negative/false-positive dilemma occurs even if the application is, for example, executed in a virtual machine, so that it cannot do permanent damage while executing.

---

[2] The curves of Figure 1 are similar to receiver operating characteristic (ROC) curves, which measure the increase in true positives with the increase in false positives. We choose false positives and false negatives for our axes because of the importance of minimizing both.

Eventually, if the application is to have any permanent beneficial effects—in fact, any effect at all, it must either update local resources (possibly compromising their integrity) or send messages to the outside world (possibly compromising the confidentiality of information). That is, the decision of whether to perform the update or send the message can be postponed, but ultimately it cannot be avoided. And that is precisely the difficult decision to make.

Given a level of knowledge of program behavior, as indicated by one of the curved lines of Figure 1, a protective system must decide whether to consider any given behavior as normal or to treat it as an attack. With a constant level of knowledge (one curve), decreasing the false negative rate necessarily implies increasing the false positive rate. With more information, the system can often decide more cases on its own. The only way to decrease both rates is to move the star toward the origin of the graph by providing more information.

Where does the additional information come from? Anomaly-based intrusion detection systems include a training period, during which a fairly complete profile is constructed, thus reducing the number of events that raise the false-negative/false-positive dilemma. What is really needed to protect new applications and releases out-of-the-box is an oracle that can distinguish appropriate from inappropriate application behavior. In the absence of such an oracle, the protection mechanism must get its information from the nearest approximation to it—namely, the application user.

There are two problems with asking advice from the application user. First, the user may not be able to provide any insight. This is the case for traditional anomaly-based application intrusion detection [Forrest96b], which profiles sequences of kernel calls made by the application. AppMon addresses this problem by profiling the application's use of resources, many of which (notably files) the user is able to answer questions about. A typical example is that the user is performing an operation that modifies a sensitive file. The user is certainly able to identity a file that he is editing and can often identify by pathname files that are important to an application.

The second problem is that interrupting the user too often to ask for advice quickly becomes annoying and then odious. AppMon addresses this problem by (a) initially reducing its goal to providing minimal protection, and (b) building a profile of the application that can represent *classes* of resources (e.g., an attachment to an email or the file being edited), about which AppMon will never need to query the user. The AppMon solution is further described in Section 6. Of course, "minimal" means minimally more restrictive than the standard access control policy, which may even include a mandatory access control (MAC) component.

1.3 AppMon focus

Malicious or incorrect software can affect the computer system on which it runs in many ways. It can compromise the *integrity* of the system by deleting files, folders, or registry keys, or altering their contents incorrectly. It can compromise the *confidentiality* of the system by sending information from the local system to another computer without authorization. It can

perpetrate a *denial of service* attack, e.g., by monopolizing computer resources to prevent other software from running.

In the Phase I effort, we have focused on what is potentially the most damaging of these situations, a compromise of integrity. In the military, highly confidential material is not apt to be stored on computers that can easily access the Internet. Denial of service is easily detected; in most cases, an alternative, uninfected computer can take up the work. Therefore, we focus in this effort on protecting the computer's integrity. Access control policies will be expressed in terms of whether the application may modify or delete a file or registry key.

1.4 Summary of Phase I accomplishments

In Phase I, we have

- Verified three key hypotheses:
  - Out of the box, AppMon can protect the integrity of system resources and critical files specified by a system administrator[3]
  - It can do so without overburdening the user
  - After customization, AppMon will allow normal behavior that a sandbox would prevent
- Developed a model of program resource use that addresses the major feasibility concerns
- Performed experiments to validate our ideas
- Defined a potential product and identified potential customers

1.5 Structure of this report

The remainder of this report is organized as follows. Section 2 briefly reviews related work. Section 3 presents a model, developed in Task 1, of how a Windows process can act on its environment. Section 5 describes a model of program execution and the algorithm we use to generate profiles of a process in terms of that model (Tasks 2 and 3). Section 6 sketches the AppMon architecture (Task 4). Section 6.3 presents our commercialization plans (Task 5) and Section 8 describes an early demo of AppMon capability. Section 9 lists the references cited in the report.

---

[3] That is, it can ensure that the application writes only to the proper resources. It cannot ensure that the modifications made to files, registry keys, etc. are correct.

2 Related work

AppMon is related to sandboxes, intrusion detection, and mandatory access control (MAC) efforts to protect computers from attack. This discussion of related work is intentionally brief. Extensive work has been done in all the research areas described.

2.1 Sandboxes

A sandbox offers a way to restrict an application's access to the host computer. The sandbox may restrict access to computer memory and scratch space on the disk. Access to the network and the local file system are often prohibited.

Examples of sandboxes include:

Applets. These are programs that are typically run by a scripting language interpreter that prevents unauthorized access to resources.

Virtual machines. A virtual machine emulates a complete host computer environment, within which the guest operating system accesses the actual machine and the network via the emulator.

By specifying limits in advance on what programs can do, sandboxing solutions inevitably prevent certain programs from executing at all. For example, an editor cannot execute in a strict sandbox, because it must have access to the file being edited. We have seen sophisticated, virtual-machine-based sandbox solutions that implement organizational policies. However, any completely predefined policy will incorrectly block some programs. In general, sandbox policies specify what is allowed, and everything else is prohibited. By contrast, an AppMon policy specifies what is *not* allowed.

2.2 Application intrusion detection

Another related area is application-level intrusion detection. Most recent work in this area builds on the pioneering work of the Forrest group at the University of New Mexico [Forrest96a, Forrest96b, Hofmeyr98, Inoue05, Warrender99]. The basic insight underlying this work is that an application may be characterized by the sequences of calls its threads make to the operating system, which is captured in a *profile*. Misbehavior by the program, or substitution of another program by a buffer overflow attack, is characterized by anomalous behavior; application intrusion detection is thus always anomaly detection.

Although later research found weaknesses in specific mechanisms proposed by the Forrest group [Tan02b, Tan02c, Wagner02], the approach has influenced all subsequent IDS research. In particular, several researchers, including the principal investigator of this effort, have proposed finite state-machine models based on the characteristic sequences of system calls [Gao04b, Marceau00, Michael02, Sekar01].

The goal of application intrusion detection is to ensure that the program that is running is in fact the program it claims to be and that it is behaving as that program normally behaves. However, this approach requires the existence of a nearly complete profile of the application; otherwise, too many false positives occur. Therefore, this approach is not able to capture life-cycle attacks that insert malicious or seriously harmful code into the application itself.

2.3 Privilege profiling

Our approach differs from that of the Forrest group in that it focuses on the application's exercise of privilege, rather than its entire interface with the kernel. Empirical privilege profiling was first suggested in [Marceau05], which shows how to build profiles of how an application uses resources. Such profiles can be used for intrusion detection, insider misuse detection, and safe execution of mobile code. A distinctive feature of our approach to privilege profiling is that it is able to abstract away from peculiarities of individual hosts, users, and sites. Thus, it is able to distinguish between use of application-specific resources (the initialization file, application registry keys) and per-execution resources (user-provided files, temporary files, etc.).

Privilege profiling avoids the over-restrictiveness of sandboxing and does not need a pre-existing profile, as application intrusion detection does. Because it is based on privilege profiling, AppMon allows most program behavior while preventing the most devastating attacks, meanwhile constructing a profile against which future behavior can be evaluated.

2.4 MAC-style access control

Mandatory access control places processes in protection domains and restricts the access of each domain. A very simple style of MAC is LOMAC [Fraser00], which protects the integrity of a computer system by defining just two domains: High and Low. High files are sensitive and may be accessed only by High processes. Any process that has been influenced by Low inputs, for example by accepting input from the Internet, becomes Low and is not allowed thereafter to write to High files. As a result, it is relatively easy to specify LOMAC domains, by contrast with SE Linux, in which many domains can be defined but domain specification is extremely difficult to get right. (It is interesting to note that Windows Vista provides something like SE Linux-style MAC to protect kernel resources only [Bradley07].)

AppMon's integrity policy is similar to LOMAC in defining just two domains, sensitive and non-sensitive. However, AppMon does not provide MAC; it is implemented above the operating system level.

3 AppMon Vision

We envision an AppMon system that works as depicted in Figure 2. The user of a not-yet-trusted application runs the application in the AppMon self-customizing monitor. In most cases, he need not be aware that the application is being monitored.

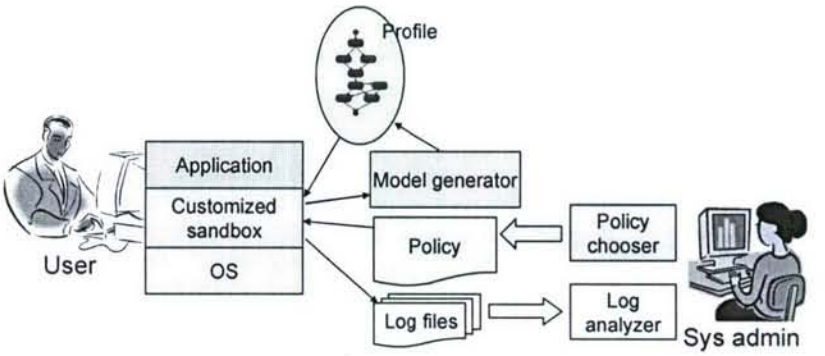

Figure 2. Conceptual view of AppMon operation

The **self-customizing monitor** is at the heart of AppMon. It checks each operation on computer resources against a profile of the application and prevents those that are clearly harmful.

Out of the box, AppMon protects OS files from modification or deletion, even if the user has administrative privilege.[4] The system administrator can increase protection by preventing the application from modifying or deleting other parts of the file system, for example, the accounting system.

As AppMon runs, it sends a stream of access information to the **profile generator**, which creates and incrementally updates a **profile** of the application's use of computer resources. AppMon in turn uses the profile to identify familiar operations, which it can then distinguish from novel behavior. The profile shows how the application uses computer resources. For example, at a certain point the application may access a user-specified file. Once AppMon expects that kind of access, it can treat those accesses differently from others, such as writes to application-specific or OS files.

By default, AppMon does not interrupt the user's work. However, it is more helpful to ask the user whether he meant to do something than to simply assume he did not.[5] For example, if a user asks the application to modify a (protected) accounting file, AppMon can check to make sure that the user knows the application is modifying a protected file. AppMon includes a graphical **user interface** to explain the situation and ask for user advice.

---

[4] On Windows, members of the Administrators group have the equivalent of root privilege on UNIX. An application invoked by such a user has complete power over the computer, a power that is regularly exploited by attackers. Eighty-five percent of Windows corporate users are in the Administrators group, in order to execute the applications they need. Many other users also run as Administrators, in order to run popular programs, such as TurboTax and many games [Chen05]. Windows Vista™ protects the integrity of the operating system and certain program files from users with administrative privilege, but does not protect enterprise data and files.

[5] For a discussion of trade-offs in interrupting the user's work, see Section 1.2.

The system administrator can control AppMon with a **policy** that governs what accesses are allowed, and when (or whether) to ask the user for advice. The **policy wizard** is an interactive program that helps the system administrator to choose a policy based on site requirements.

The stream of application actions sent to the profile generator is also logged for possible later analysis by a system administrator or other authorized person, using a **log analyzer. The log analyzer** helps its user navigate the log and identifies entities of interest, such as protected files, successful writes, and unsuccessful attempts to modify system resources.

4 A model of application interaction with the Windows environment (Task 1)

Excluding denial of service (e.g., crashing), a program can do harm in any of several ways:

(1) It can directly violate the integrity or confidentiality of computer resources, e.g., modifying the registry at the behest of an intruder or sending the contents of a local file to an IP address specified by an attacker

(2) It can invoke a Windows service inappropriately, for example to create a new user (a remote attacker)

(3) It can ask another application to do (1) or (2) or to cause other harm—e.g., by sending email or spam purporting to come from the logged-in user

(4) It may incorrectly alter or delete a shared resource that is needed by other applications (for example, by "upgrading" a file for which another application needs the older version)

(5) It can compute its results incorrectly

(6) Even if its computations are correct, it may be given incorrect input and produce incorrect results

(7) It can impede operation (denial of service), e.g. by killing one or more processes

AppMon targets the first three of these damage modes. It does not attempt to detect computation of incorrect results or denial of service ((5) through (7)). In Phase I, we addressed threats to integrity but not to confidentiality.

AppMon does not attempt to thwart all possible attacks by sophisticated villains, possibly aided by local collaborators. Therefore, it provides less protection than, for example, applet sandbox environments or mandatory access control. Rather, AppMon guards against the major ways in which integrity or confidentiality of computer resources may be compromised, as shown in Figure 3: reading or writing to files, reading or writing to registry keys, Internet communications (e.g., transmitting local information), and using other applications to perform those actions (Figure 3 shows an application using a mailer to send information to an attacker). In addition, the application may call on operating system services to cause harm; one example is to create a new login identity that can be used in another stage of an attack.

The protection provided by AppMon can be roughly stated as follows: it provides a higher level of access control than provided by DAC (by specifying access per-application, in addition to

DAC's per-user) but without the cost (or benefits) of MAC. Its control is at the granularity of entire resources; thus it can prevent an application from writing bad data to certain precious files, but does not prevent harm to unprotected files. For those resources that are protected, however, AppMon will prevent harm via third-party applications—for example, passing the name of a file to another application for modification.

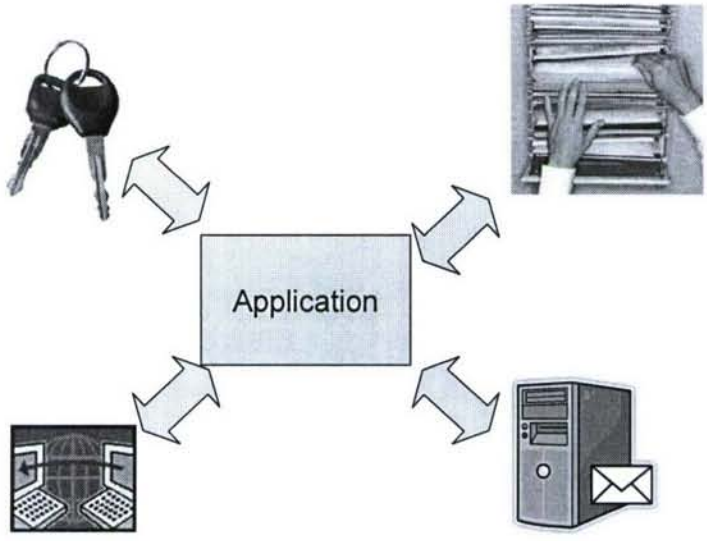

Figure 3. An application can cause harm via files, registry keys, the Internet, or other applications

We discuss particulars of the different types of resources in the remaining subsections of this section. (In Phase I, we have focused on files, since they are a common target.) In this report, we do not discuss in-memory resources, such as the active registry state.

4.1 Computer resources (files and registry keys)

The most obvious way in which an attack can cause harm is to compromise either the integrity or the confidentiality of information in the file system or in the registry keys, which are accessible through the API of the kernel, which is defined in a dynamic link library (DLL) called ntdll.dll. AppMon therefore monitors registry key and file operations. Although it is not clear how some ntdll operations can cause harm (e.g., NtUnlockFile appears harmless), we monitor all of the operations in order to track the current state of the application thread.

Reads from and writes to files and registry keys can be captured at the level of calls to the kernel interface, ntdll.dll. Various types of instrumentation have been developed—including at ACT-NY—to capture calls to ntdll.dll operations.

AppMon captures all file and registry key operations, as shown in Table 1. Most operations require access rights to the file or key.

Ntdll.dll file operations	Ntdll.dll registry key operations
NtCreateFile	NtCreateKey
NtOpenFile	NtOpenKey
NtReadFile	NtDeleteKey
NtWriteFile	NtFlushKey
NtQueryInformationFile	NtSaveKey
NtSetInformationFile	NtRestoreKey
NtQueryDirectoryFile	NtSaveMergedKeys
NtQueryEaFile	NtLoadKey
NtSetEaFile	NtLoadKey2
NtQueryAttributesFile	NtUnloadKey
NtQueryFullAttributesFile	NtQueryOpenSubKeys
NtQueryDirectoryFile	NtReplaceKey
NtLockFile	NtSetInformationKey
NtUnlockFile	NtQueryKey
NtDeleteFile	NtQueryValueKey
	NtEnumerateKey
	NtNotifyChangeKey
	NtNotifyChangeMultipleKeys
	NtDeleteValueKey
	NtSetValueKey
	NtEnumerateValueKey
	NtQueryMultipleValueKey
	NtInitializeRegistry

Table 1. Ntdll.dll file and registry key operations

4.2 Use of the network or Internet

Threats to confidentiality involve reading and then exporting confidential information. Exporting the information can occur by writing it to a file (e.g., an innocuous looking file that may be harvested at a later time), sending it over a network, or handing it off to another application (which may write it to a file or send it over the network).

In Windows, communication with the network does not take place through ntdll.dll. Any process may interact directly with networking software in Winsock [Winsock], which supports TCP/IP communications. Very few applications use any other low-level protocols. However, an attack might deliberately use a rare protocol in order to escape detection. The intention is that AppMon will monitor the application's use of Winsock (TCP/IP) and simply note the use of other, obscure protocols.

AppMon can note the use of connect() calls to connect to other computers over the Internet and bind() calls to accept incoming calls. For example, the AppMon profile for an application might record the fact that the application always connects to a certain port at an Internet address belonging to the application vendor. The profile of a Web browser would include the fact that the browser connects to a variety of Web servers (port 80).

4.3 Other applications

Many applications invoke other applications. For example, Outlook invokes other applications to open attachments; Word invokes appropriate applications to handle embedded documents. A danger in such situations is that a compromised application hands a sensitive document to another application to be modified inappropriately or exposed to unauthorized persons.

AppMon will handle this situation by monitoring the execution of the invoked application in the same way as the original application. The invoked application will be treated as novel or unfamiliar code; hence, it will be constrained at least as much as the first application.

4.4 Services

Another major way in which an application can threaten the confidentiality or integrity of a Windows system is by invoking system services. Although in general these services do not directly modify or delete data, they are an extension of the operating system. For example, the Local Security Authority manages security on the local computer; it creates new users and controls file encryption.

At the beginning of this effort, we thought it might be possible to capture communication with services at the ntdll.dll level—that is, capture messages via pipe to system services. In studying the services, however, we found that this is infeasible. The message formats are internal to Microsoft and subject to change. Therefore, we intend to capture these at a higher level than ntdll.dll (the kernel API).

Capturing such calls at a high level—e.g., a call to kernel32.dll, advapi32.dll or some other OS DLL—means that a sophisticated attacker could in principle craft the remote procedure call (RPC) to the system service that the OS DLL would make. Currently, attackers do not need to do so, although there is little doubt that they could. At best, then, AppMon raises the bar for would-be attackers, carrying the ongoing arms race to the next level.

5 A model of program execution (Tasks 2 and 3)

In this section, we describe our model of program execution and a method for generating profiles based on that model. From the profile, AppMon will be able to distinguish between resources used in the normal way by the program and usage that has not been encountered before.

In earlier work on resource profiling, we developed a technique for identifying the resources used by an application – e.g., the initialization file, the log file, the file being edited. This method works even when files, such as the file being edited, are different in different executions. This method is effective but the instrumentation technique is much more complicated than simply instrumenting calls made to ntdll operations.[6] Hence, in Phase I of this effort, we have developed a new way to profile an application's use of resources and a technique to generate the alternative profiles. This technique of Phase I is based only on sequences of calls by the application to the kernel, through the library that defines the kernel API, ntdll.dll. In experiments with several applications, this method has produced useful profiles. The profile is based on two aspects of resource use: (1) the *sequence of operations* on a given resource and (2) *resource clusters*.

Sequence of operations. In considering operations on a given resource, we include not only the operation name but also some other parameters of the operation – for `NtCreateFile`, for example, we include the requested access rights and the `CreateDisposition` parameter.[7] These parameters alone often distinguish between the program's use of distinct resources. An example of a sequence of operations is shown in Figure 4. (In this figure, FILE_CREATE_READ_CLOSE stands for a sequence of ntdll operations—in this case: NtCreateFile, NtReadFile, and NtClose.)

```
c:\texmf\tex\latex\graphics
   \dvips.def
      3948: [1][1] FILE_READ
      3948: [1][3] FILE_CREATE_READ_CLOSE
      3948: [1][1] FILE_READ
      3948: [1][5] FILE_CREATE_READ_CLOSE
   \color.sty
      3948: [1][1] FILE_READ
      3948: [1][3] FILE_CREATE_READ_CLOSE
      3948: [1][1] FILE_READ
      3948: [1][5] FILE_CREATE_READ_CLOSE
   \dvipsnam.def
      3948: [1][1] FILE_READ
      3948: [1][3] FILE_CREATE_READ_CLOSE
      3948: [1][1] FILE_READ
      3948: [1][5] FILE_CREATE_READ_CLOSE
```

Figure 4. Sequence of operations on a LaTex file

---

[6] We previously reported that the earlier method was not applicable to applications written in interpreted languages. Since then, we have discovered how to apply it successfully to such programs. However, the instrumentation method itself remains somewhat fragile.

[7] The `NtCreateFile` operation combines six distinct operations; the `CreateDisposition` parameter identifies which of the six the program is invoking.

Resource clusters. Both files and registry keys have hierarchical pathnames. In many cases, two distinct resources used by a program are related (clustered) by their names. For example, an editor uses "the file to be edited" and "the directory in which the file is located." These two form a cluster, because the first (the file) is always located in the second (the folder). In programs that we have observed to date, the cluster relationships, together with the sequences of operations, are sufficient to identify particular resources used by the program.

Thus, the profiles of Phase I may be viewed as a cross-product of state machines, each of which tracks operations on a single resource and which are related (clustered) by name.

At the completion of Phase I, we are poised to create profiles for a wide range of applications and testing to ensure their ability to discriminate between harmful and benign application behavior. We note, however, that the final demo presented in March and described in Section 7.2 made use of the old-style profiles rather than the new application profiles.

6 AppMon architecture (Task 4)

The overall structure we envision for AppMon was described in Section 3. In this section, we describe in more detail the access policies that it uses to protect the host computer. We then discuss how AppMon learns about the application over time. Finally, we tackle the difficult issue of responding to possibly harmful events.

6.1 Access policies

In this section, we discuss the policies that govern AppMon. We have developed a model policy for protecting system *integrity*. Policies to protect confidentiality are part of future work.

6.1.1 The core access policy

We assume that an application, instrumented with AppMon, is running on one or more computers in an organization. Each execution of the application produces a log file; log files are automatically composed to generate a composite profile of the application's use of resources.

Before a profile has been generated, AppMon enforces a *core* access policy. This access enforcement is in addition to the normal Discretionary Access Control (DAC) enforced by the operating system. Normal DAC controls access by *users* and groups of users to computer resources. AppMon controls the application's access to resources. Therefore, even if a user has access to many critical files, for example, on his computer, the application will be able to access them only if AppMon's policy allows it to.

The purpose of the core access policy is to protect the computer from the worst attacks or errors during training. During training, AppMon develops a profile of the application's normal use of

resources. After training is completed, a more nuanced policy can be used, which both tightens access to application resources and enables the user to access even protected files if necessary.

6.1.2 Domain/type access policies

AppMon access policies are based on a domain/type access model. That is, we define domains in which the application may execute and types (sets) of resources. We then specify the access that each domain has to each type of resource. Then to make an access decision, it is necessary to know (a) the domain that is executing and (b) the type of the resource.

Resource types are specified implicitly. For example, we say that every file whose absolute pathname begins with `c:\system\system32` has type `system_t`[8] (system files). More precisely, each type is defined by a set of pathnames, each of which defines a subtree of the file system. A file has the type associated with the longest match to the file's absolute pathname. AppMon predefines five types. In principle, users can define additional types if needed. The types are:

Windows_t. This is the default resource type. Any resource not specified to be of some other type is of this type.

System_t. System resources are specified by Microsoft. System files and registry keys are known in advance by pathname.

Protected_t. The user or system administrator can (optionally) define some resources to be protected. For example, he might protect accounting data when a not-yet-trusted editor or IDE is run.

App_t. These resources are needed by the application; they are typically created as part of application installation.

AppMon predefines five domains. In principle, more domains can be defined, if users wish to do so. The domains are:

Unknown_d. The application is executing in a state not yet profiled by AppMon.

System_d. The application is executing in a state in which it accesses a system resource (i.e., a resource in system_t, a resource defined by the operating system).

App_d. The application is executing in a state in which it accesses an application resource.

Process_d. The application is executing in a state in which it accesses arbitrary or per-process resources, such as a file specified by the user or a temp file.

Extended_d. A process invoked by the application is executing, possibly with data or arguments provided by the application.

---

[8] We follow common practice in giving types names that end in "`_t`" and domains names that end in "`_d`".

6.1.3 Integrity protection core policy

The core policy protects the computer from software about which nothing is known. When a user first begins to run an instrumented application, the only domain is unknown_d. The only known types are `system_t`, `windows_t`, and `protected_t`.

We can represent the policy by the matrix of Figure 5. In this figure, YES means that the resource can be modified or deleted, while NO means that it cannot.

Domain	System_t	Windows_t	Protected_t
Unknown_d	NO	YES	NO

Figure 5. Core integrity-protection access matrix

The matrix of Figure 5 simply says that the application cannot modify or delete system or protected files or registry keys, even if the user normally has the right to do so. For all other files and registry keys, normal DAC applies.

6.1.4 The AppMon policy after training

If we knew all the domains and types for the application, we could use the access policy matrix of Figure 6. In this section, we first explain the matrix and then argue the merits of the policy it represents. In the next section, we will explain how AppMon discovers the domains and types it needs in order to use the matrix.

Domain	System_t	App_t	Windows_t	Protected_t
System_d	YES	NO	NO	NO
App_d	NO	YES	NO	NO
Process_d	?	?	YES	?
Extended_d	NO	NO	YES	NO
Unknown_d	NO	?	YES	NO

Figure 6. Core integrity-protection access matrix, all types and domains

As in Figure 5, the cells of Figure 6 specify whether the application may modify or delete a resource. This is because we are concerned only with system integrity. A more general policy would specify both read and write permissions.

The access matrix of Figure 6 introduces a new cell value: "?" The question mark indicates that the policy does not know whether the attempted modification should be allowed. In such a case, AppMon can make one of two responses (configurable): it can ask the user or simply deny the request.

6.1.5 Asking the user for advice

Asking the user for help with security issues is usually a bad approach. However, we argue that it will work here. The problems with asking the user are that (a) it imposes an onerous burden on the user, who simply wants to get his work done; and (b) the user never knows the answer to the question anyway.

We first address the question of burden. We have used this approach in constructing resource usage profiles of Outlook, an example of an extremely complex application. When fourteen users ran Outlook, they used almost 70 application files (many of which were DLLs). To distinguish between application and per-process files (essentially to identify App_t directories), we asked the user about probable application files and directories. A total of 15 questions to the user were required. If Outlook required only 15 questions, most applications will require fewer. Therefore, we argue that these questions do not pose too great a burden for the user.

We also claim that the user is able to answer questions about files on the computer. In our Outlook example, most users can tell that a file in a directory named "outlook" is related to the application, while a spam filter is not.

6.1.6 Summary: advantages of the AppMon approach

The AppMon approach protects the computer while ensuring that the user can get his work done and remaining easy to use:

- AppMon restricts the rights of the application without restricting the rights of the user. This is true even if the user is an Administrator or otherwise enjoys extensive DAC privileges. No new pseudo-users have to be created and their access rights to computer resources correctly determined.
- When new, untried code is executed, AppMon prevents harm to the operating system and to resources that the user (or system administrator) deems critical.
- When a familiar part of the application is executed, AppMon relaxes restrictions on execution, so that the user can modify resources as necessary. In this case, however, it still asks the user about system or protected files.

6.2 Development of the core integrity policy

The access matrix of Figure 6 represents the access policy that AppMon uses to ensure system integrity when it knows the application states in System_d, App_d, and Process_d, and the resources in App_t. However, during training those states and resources are not known. In this section, we briefly describe how the access matrix of Figure 6 is filled in during training.

Figure 7 shows the access matrix of Figure 6, but with most cells left white. The white cells for domains System_d, App_d, and Process_d indicate that when training begins, there are no known states in those domains. The white cells in the column for type App_t indicate that when

training begins, no resources of type App_t are known. It is the job of AppMon to discover the states in those domains and the resources of that type.

The only domains that are relevant during training are Unknown_d, which includes all states in the application, and Extended_d, which includes all states in other applications invoked by the given application.

Domain	System_t	App_t	Windows_t	Protected_t
System_d	YES	NO	NO	NO
App_d	NO	YES	NO	NO
Process_d	?	?	YES	?
Extended_d	NO	NO	YES	NO
Unknown_d	NO	?	YES	NO

Figure 7. AppMon access policy when training begins

As AppMon runs, it notices that certain states of the application always refer to the same resource. For example, at a certain point the application always reads its initialization file. (AppMon does not know that the file is the initialization file; it merely knows that it is always the same file.) As a result, AppMon is able to infer that that file belongs to the application, i.e., it belongs to App_t.

As AppMon learns application states and the resources they use, it can fill in the App_d row of the access matrix. In addition, since some files of App_t are now known, more of the App_t column can be filled in, as shown in Figure 8.

Domain	System_t	App_t	Windows_t	Protected_t
System_d	YES	NO	NO	NO
App_d	NO	YES	NO	NO
Process_d	?	?	YES	?
Extended_d	NO	NO	YES	NO
Unknown_d	NO	?	YES	NO

Figure 8. Access policy when App_d is known

At the same time, AppMon notices that other places in the code access many different resources; they never seem to read or write the same file twice. These are the states in

Process_d, which write to temp files or user-specified files. As these states become known, the row of the access matrix for Process_d is filled in.

Domain	System_t	App_t	Windows_t	Protected_t
System_d	YES	NO	NO	NO
App_d	NO	YES	NO	NO
Process_d	?	?	YES	?
Extended_d	NO	NO	YES	NO
Unknown_d	NO	?	YES	NO

Figure 9. Complete access control matrix

Note that, given the policy of Figure 6 through Figure 9, the `System_d` row of the access matrix will never be filled in. That is because the core policy of Figure 6 does not permit writes to system files. Therefore, none will occur, and no system states will be discovered. If the application indeed needs to write to system files or registry keys, this will be a disadvantage of the policy. However, applications that need to write to system files are rare; we have profiled Outlook's use of resources for another research effort, and discovered no instances in which system DLLs invoked by Outlook modified system files.

6.3 Response

AppMon's response to anomalies must be configurable and almost completely automated. Specifically,

- The system administrator should be able to configure the response, requesting more or less control. (However, it can't ask AppMon to act with human intelligence.)
- AppMon should be prepared to act automatically
- The user should have ultimate control over any response that denies him service.

Configurability. Possible responses include asking the user's advice, stopping the application, and pausing execution while an expert is consulted. Automatic response may be conditional (e.g., stop the application only if a highly sensitive resource is involved); the system administrator must be able to set the parameters.

Automatic response. AppMon cannot take the often useful approach of slowing down the computer [Somayaji00]. Because it works with attempts to access resources, a single anomalous event may be fatal. Instead, AppMon will have to gauge the potential seriousness of an anomaly based on the affected resource. Anomalous attempts to modify system files, for example, are highly suspicious. AppMon can accomplish more with some hints from a system

administrator about the sensitivity of resources, such as files (directory hierarchy sub-trees), registry keys, pipes to other applications, TCP ports, and IP addresses.

User control. Interactive applications should consult the user before interrupting his work.

7 Commercialization

In this section, we first present requirements and our commercialization contacts for AppMon. Finally, we describe a demonstration of AppMon capability suitable for showing to potential commercial partners. The demo was prepared during Phase I and presented to the Air Force in March 2007.

7.1 Requirements

We have developed a list of AppMon stakeholders and requirements on AppMon from their various points of view. This list is summarized in Table 2.

Stakeholder	Description	Requirements
Application user	Uses application	(1) AppMon should be silent and invisible, with minimal or no performance impact
DISA	Responsible to ensure that military computers are defended against enemies who threaten data integrity, confidentiality, and availability	(2) Solutions to the problem of running new software safely may need DISA approval
		(3) AppMon must protect against attacks through the application, including zero-day and life cycle attacks, with low miss rate (false negatives)
		(4) The subset of attacks prevented by AppMon must be significant and include most probable attacks
Sys admin	Must install and maintain AppMon	(5) AppMon must be easy to install and update
Responder	Responds to alerts generated by AppMon	(6) AppMon must support correct response; it cannot assume users and system administrators with significant security training
		(7) AppMon's false positive rate must be close to zero.
Patch-management service provider	Maintains and updates application patches on computer	(8) Service provider must be aware that application is run with AppMon and must be able to obtain and wrap new versions of application (Note: this stakeholder may not exist for all systems)

Table 2. AppMon stakeholders and requirements

7.2 Commercialization contacts

We have contacted Cyber Security Technologies and presented AppMon's capabilities and potential to them. The flagship product of Cyber Security Technologies is the OnLine Digital Forensic Suite™, an innovative computer forensic toolset that is gaining popularity with law enforcement as well as corporate investigators. Cyber Security Technologies is particularly interested in AppMon's ability to record resource usage and to flag anomalous or suspicious accesses.

8 Demonstration of AppMon for potential commercialization partners

To attract commercialization partners, it is crucial to have early prototypes of a product's operation. To demonstrate how AppMon will work, we simulated AppMon monitoring three applications, as depicted in Figure 10. Although the demo uses the technology of [Marceau05] rather than that being developed for AppMon (see Section 5), it provides a reasonable idea of AppMon operation to potential and actual commercialization partners.

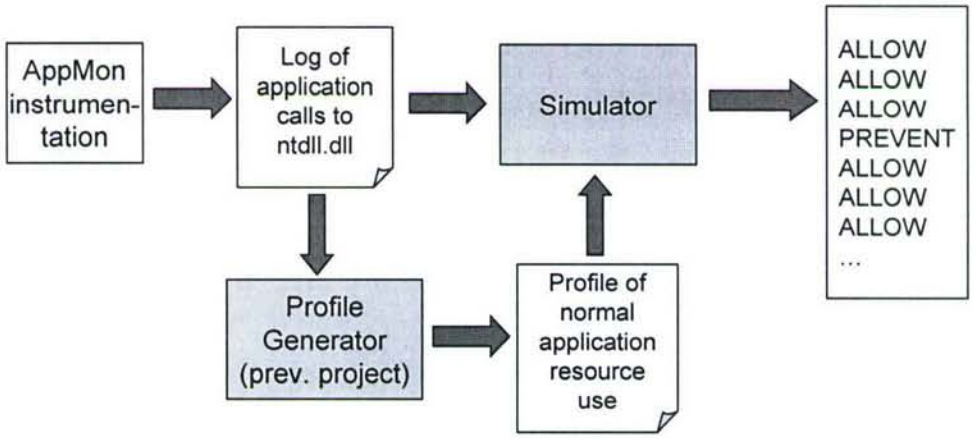

Figure 10. AppMon demo

Using **AppMon instrumentation**,[9] we logged the application's use of resources, at the ntdll operation level. Based on several logs and using a **profile generator** from a previous project, we created a **profile** of the application's use of resources. (The implementation of AppMon profiles is not yet mature enough to support anomaly detection.)

The simulator reads logs and simulates the action of AppMon:

- Either out of the box or with a profile available. Logs and profiles are available for four applications: Microsoft Calculator, Microsoft Outlook, Microsoft Notepad, and an application of our own creation, called minigrep, that is capable of attack-like behavior. (Minigrep was developed at ATC-NY for the purpose of this demo, in order to have an example of attack behavior.)
- Based on a policy. Two policies are built in. The default ("user control") policy asks the user when a suspicious write is attempted. The alternative ("no interruptions") policy can also be selected.
- The simulation addresses only files, since the profiles developed for the previous project describe only file use.

---

[9] Words and phrases in bold refer to elements of Figure 10.

The simulation demonstrates three characteristics of AppMon that are critical for its success.

First, out of the box—that is, without a profile of the application, **AppMon can protect the integrity of critical files**. Critical files include system files (those on which the operating system relies) and protected files (those that a system administrator deems critical to the mission and that application should not have to modify or delete). This was shown in two ways:

- AppMon detects and can prevent the Microsoft Calculator's attempts to write to the Microsoft initialization file.
- AppMon detects and can prevent minigrep's attempts to write to protected files (if the "`C:\program files`" subtree is protected).
- AppMon detects and prevents an attempt by the attack embedded in minigrep from writing to a system file.

Second, AppMon protects the integrity of the computer **without bothering the user**. Even when AppMon is configured (by policy) to ask the user about suspicious write attempts, interruptions are rare. This is because attempts to write system files are very rare and, if the system administrator chooses files to protect wisely, attempts to write to protected files will be very rare.

- The log of a lengthy execution of Outlook produced no writes to system files at all. In fact, a profile created from fifteen executions of Outlook by fifteen different users contains no writes to system files. This corroborates our claim that writes to system files are rare, since Outlook is well-known to be a complex program.

Finally, after customization, **AppMon will allow normal behavior that a sandbox would prevent**.

- The log of the minigrep execution records that output was directed to a file in the `C:\program files\` subtree. When this subtree was protected, AppMon prevented the write before customization, as noted above. However, after customization, when the profile of minigrep was available, AppMon allowed the write to occur.

9 Conclusion

The Phase I effort has shown that the AppMon approach to protecting computers from new software applications, patches, and releases is feasible. AppMon offers a reasonably high level of protection for the most important computer resources, preventing many kinds of errors and effectively "raising the bar" for would-be attackers. A Phase II effort will be required to turn the AppMon approach of Phase I into a product prototype.

10 References

[Bradley07] T. Bradley, Introduction to Windows Integrity Control,

[Chen05] S. Chen, J. Dunagan, C. Verbowski and Y.-M. Wang, "A Black-Box Tracing Technique to Identify Causes of Least-Privilege Incompatibilities," in *Proceedings of NDSS 2005*, 2005.
[Forrest96a] S. Forrest, S. Hofmeyr and A. Somayaji, "Computer Immunology," in *Communications of the ACM* 40 (10), 1997.
[Forrest96b] S. Forrest, S. A. Hofmeyr and A. Somajayi, "A Sense of Self for UNIX Processes," in *Proceedings of IEEE Symposium on Computer Security and Privacy*, 1996.
[Fraser00] T. Fraser, "LOMAC: Low Water-Mark Integrity Protection for COTS Environments," in *Proceedings of IEEE Symposium on Security and Privacy*, 2000.
[Gao04b] D. Gao, M. K. Reiter and D. Song, "On Gray-Box Program Tracking for Anomaly Detection," in *Proceedings of USENIX Security Symposium*, 2004, pp. 103-118.
[Hofmeyr98] S. A. Hofmeyr, S. Forrest and A. Somayaji, "Intrusion detection using sequences of system calls," in *Journal of Computer Security* 6 (3), pp. 151–180, 1998.
[Inoue05] H. Inoue, *Anomaly Intrusion Detection in Dynamic Execution Environments*, Ph.D. Thesis, Computer Science, University of New Mexico, 2005.
[Marceau00] C. Marceau, "Characterizing the Behavior of a Program Using Multiple-Length N-grams," in *Proceedings of New Security Paradigms Workshop*, 2000.
[Marceau05] C. Marceau and R. Joyce, "Empirical Privilege Profiling," in *Proceedings of New Security Paradigms Workshop*, 2005.
[Michael02] C. C. Michael and A. Ghosh, "Simple, State-based approaches to program-based intrusion detection," in *ACM Transactions on Information and System Security* 5 (3), pp. 203-237, 2002.
[Sekar01] R. Sekar, M. Bendre, D. Dhurjati and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings of IEEE Symposium on Security and Privacy*, 2001, pp. 144-155.
[SELinux] National Security Agency, Security-Enhanced Linux, http://www.nsa.gov/selinux/.
[Somayaji00] A. Somayaji and S. Forrest, "Automated Response Using System-Call Delays," in *Proceedings of 9th USENIX Security Symposium*, 2000.
[Tan02b] K. M. C. Tan, K. S. Killourhy and R. A. Maxion, "Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits," in *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.
[Tan02c] K. M. C. Tan and R. A. Maxion, "Why 6? Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector," in *Proceedings of IEEE Conference on Security and Privacy*, 2002, pp. 188-201.
[Wagner02] D. Wagner and P. Soto, "Mimicry attacks on host based intrusion detection systems," in *Proceedings of Ninth ACM Conference on Computer and Communications Security*, 2002.
[Warrender99] C. Warrender, S. Forrest and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *Proceedings of IEEE Symposium on Security and Privacy*, 1999, pp. 133-145.
[Winsock] Microsoft (TM), Winsock Reference, http://msdn2.microsoft.com/en-us/library/ms741416.aspx.